ROUNDING OPERATIONS IN COMPUTER PROCESSOR

INVENTORS: FRANS W. SIJSTERMANS AND JOS VAN EIJNDHOVEN

Technical Field

[0001] The invention relates to programmable processors and, more particularly, to mathematical operations in such processors.

Background

[0002] In conventional programmable processors, floating point operations are mathematical operations in which the operands and the results are represented in fractional form. In certain situations, however, it is desirable to obtain an integer result. For example, while the floating point operation 15/4 yields the result 3.75, it may be preferable under some circumstances to return a result of 3 or 4. In such cases, the preferred result can be obtained using a rounding operation.

[0003] Some processors perform such rounding operations by rounding down in all cases or, alternatively, by rounding up in all cases. For processors that round down in all cases, for example, a value of 3.999 rounds to 3, while a value of –3.999 rounds to –4. By contrast, for processors that always round up, a value of 3.999 rounds to 4, while a value of –3.999 rounds to –3. While these results are acceptable for some applications, other applications, such as those intended to be compliant with standards, require symmetrical rounding. For such applications, for instance, if a value of 3.999 rounds to 3, a value of –3.999 should round to –3, rather than –4.

[0004] Some approaches for obtaining compliant rounding results in these situations involve the use of greater degrees of precision. For example, to obtain 16-bit compliance with standards, some processors use 32-bit precision. This approach can

produce compliant rounding results, but performance is typically compromised due to the additional operations involved with the higher degree of precision.

[0005] Accordingly, a need continues to exist for an integer rounding technique that will produce results that comply with standards, while maintaining efficient processor performance.

Summary

[0006] According to various implementations of the invention, rounding operations are performed by machine instructions based on a selectable rounding mode. A machine instruction is used to set the rounding mode, which is automatically applied to subsequent arithmetic operations. Using machine instructions to round results according to the selected rounding mode has several advantages over software-implemented rounding techniques, such as improved execution speed and increased conciseness of code.

[0007] A variety of rounding modes are available for selection using, for example, a three-bit rounding mode field of a control register. A rounding term is added to a result of a floating-point operation, depending in part on the selected rounding mode and on a sign (+ or -) of the result. Adding the rounding term ensures that the desired rounding operation can easily be obtained by a right-shift operation regardless of the selected rounding mode.

[0008] In one embodiment, the invention is directed to a method in which a first instruction is executed in a programmable processor to set a rounding mode. A second instruction is executed within the programmable processor to generate an integer result rounded according to the rounding mode.

[0009] In another embodiment, the invention is directed to a method in which an operation is performed and the result is rounded according to a selectable rounding mode. A rounding term is added to a result of the integer operation to obtain an intermediate result. The rounding term is determined at least in part as a function of the rounding mode, a shift amount, and a sign of the result of the operation. The intermediate result is then right-shifted by the shift amount.

[0010] Other embodiments of the invention include methods for compiling programs for performing these methods, as well as computer-readable media and apparatuses for performing these methods. The above summary of the invention is not intended to describe every embodiment of the invention. The details of one or more embodiments of the invention are set forth in the accompanying drawings and the description below. Other features, objects, and advantages of the invention will be apparent from the description and drawings, and from the claims.

Brief Description of the Drawings

[0011] Figure 1 is a block diagram illustrating an example programmable processor configured to perform rounding operations consistent with the principles of the invention.

[0012] Figures 2-15 are flow diagrams illustrating example modes of operation of the programmable processor methods.

[0013] Figure 16 is a block diagram illustrating the programmable processor within a computing system.

Detailed Description

[0014] In general, the invention facilitates rounding operations within a programmable processor in accordance with a selectable rounding mode. A single machine instruction can be used to select one of a variety of rounding modes. Some example rounding modes that can be selected include, but are not limited to, rounding up (or down) in all cases, rounding toward (or away from) zero in all cases, and rounding to the nearest integer. In the latter mode, a rule for handling a fractional part equal to 0.5, *e.g.*, how to round the value 3.5, can also be specified as described more fully below.

[0015] Once a rounding mode is set, the programmable processor uses the mode when executing subsequent machine instructions that perform arithmetic operations. The mode may remain in effect until it is changed. Using machine instructions in this way, rather than software-implemented rounding, realizes a number of advantages, including faster execution and more concise code.

[0016] In this detailed description, reference is made to the accompanying drawings that form a part hereof, and in which are shown by way of illustration specific embodiments in which the invention may be practiced. It is understood that other embodiments can be utilized and structural changes can be made without departing from the scope of the invention.

[0017] Figure 1 is a block diagram illustrating a programmable processor 10 arranged to support rounding operations in a manner consistent with the principles of the invention. The description of Figure 1 is intended to provide a brief, general description of suitable processor hardware and a suitable processing environment with which the invention may be implemented. Although not required, the invention is described in the general context of instructions being executed by processor 10.

[0018] In accordance with the invention, processor 10 supports an extensive set of arithmetic operations such as, for example, arithmetic operations, including addition, subtraction, and multiplication; logical operations, such as logical AND and OR operations; comparisons; and a variety of multimedia operations. All of these operations are supported for byte, half word, and word vectors and may be performed on integer or floating-point operands. In addition, some operations are also supported for double word vectors.

[0019] If a result of an integer operation is not representable, the bit pattern that is returned is operation-specific, as described more fully below in connection with the descriptions of example integer operations. Generally, the returned bit pattern for regular addition and subtraction operations is a wrap around bit pattern. For DSP-type operations, the returned bit pattern is typically clipped against the minimum or maximum representable value. Operations that produce a double precision result, such as integer multiplication, can return the least significant half of the double precision result. Alternatively, the most significant part of the double precision result can be returned, possibly after rounding.

[0020] As shown in Figure 1, processor 10 includes control unit 12 coupled to one or more functional units 14. Control unit 12 controls the flow of instructions and/or data through functional units 14. For example, during the processing of an instruction, control unit 12 directs the various components of processor 10 to fetch and decode the instructions, and to correctly perform the corresponding operations using, for example, functional units 14. Additional units such as fetch unit 16, decode unit 18, or a decompression unit may be coupled to functional units 14 and controlled by control unit

12. In addition, functional units 14 are also coupled to a register file 20, which stores both the operands and the results of operations. Control unit 12 includes control register 22, which stores an indicator of the particular rounding mode to be applied in subsequent arithmetic operations.

[0021] In some implementations, the functional units 14 are pipelined such that operations can be loaded into a first stage of a pipelined functional unit and processed through subsequent stages. A stage processes concurrently with the other stages. Data passes between the stages in the pipelined functional units during a cycle of the system. The results of the operations emerge at the end of the pipelined functional units in rapid succession. In other implementations, the functional units 14 are not pipelined.

[0022] Though not required, in one mode of operation, the fetch unit 16 fetches an instruction from an instruction stream. This instruction is then decoded by decode unit 18, and delegated to the appropriate functional unit 14 by control unit 12. The functional unit 14 retrieves the operand or operands from the register file 20, executes the instruction according to the rounding mode specified by the control register 22, and writes the result of the operation into the register file 20.

[0023] The methods and techniques described herein can be implemented in connection with a variety of different processors. For example, the processor 10 can be any of a variety of processor types, such as a reduced instruction set computing (RISC) processor, a complex instruction set computing (CISC) processor, variations of conventional RISC processors or CISC processors, or a very long instruction word (VLIW) processor. The VLIW architecture may include a plurality of instruction slots each having an associated set of functional units 14, and each slot may be adapted to

execute one operation of a VLIW instruction. While each slot can have an associated set of functional units 14, only one functional unit 14 in a given slot can be used at any given time.

[0024] In some implementations, the VLIW processor allows issue of five operations in each clock cycle according to a set of specific issue rules. The issue rules impose issue time constraints and a result writeback constraint. Issue time constraints result because each operation implies a need for a particular type of functional unit. Accordingly, each operation requires an issue slot that has an instance of the appropriate functional unit type attached. These functional units require time to recover after performing an operation, and during this recovery time, other operations that require a functional unit that is being recovered cannot be performed. Writeback constraints result because no more than five results should be simultaneously written to the register file 20 at any point in time. Any set of operations that meets the issue time and result writeback constraints constitutes a legal instruction.

[0025] By way of example, some details of the invention will be described in the context of a VLIW processor. It should be noted, however, that the invention is not limited in implementation to any particular type of processor, and any description of a particular processor type should not be construed to limit the scope of the invention.

[0026] Processor 10 typically includes or is used in conjunction with some form of processor readable media. By way of example, and not limitation, processor readable media may comprise computer storage media and/or communication media. Computer storage media includes volatile and nonvolatile, removable and nonremovable media implemented in any method or technology for storage of information such as processor-

readable instructions, data structures, program modules, or other data. Computer storage media includes, but is not limited to, random access memory (RAM), read-only memory (ROM), EEPROM, flash memory, CD-ROM, digital versatile discs (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium that can be used to store the desired information and that can be accessed by the processor 10. Communication media typically embodies processor readable instructions, data structures, program modules, or other data in a modulated data signal, such as a carrier wave or other transport medium and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media, such as a wired network or direct-wired connection, and wireless media, such as acoustic, RF, infrared, and other wireless media. Computer readable media may also include combinations of any of the media described above.

[0027] Figure 2 depicts an example mode of operation 200 of processor 10 for performing a shift right operation, according to a particular embodiment of the invention. The shift right operation is typically used to implement integer division by some power of two. For example, shifting three bits to the right has the effect of dividing the integer by eight. The conventional shift right operation rounds down, *i.e.*, toward negative infinity. According to the invention, this conventional operation is supported. In addition, however, the user can select from a variety of alternative rounding modes by setting an integer rounding mode field of a Program Control and Status Word (PCSW) register stored, for example, in the control register 22 of Figure 1. Assuming the integer rounding

mode field contains three bits, eight distinct integer rounding modes are selectable. These rounding modes include:

- rounding down (toward negative infinity),
- rounding up (toward infinity),
- rounding to the nearest integer, with 0.5 being rounded toward negative infinity,
- rounding to the nearest integer, with 0.5 being rounded toward infinity,
- rounding to the nearest integer, with 0.5 being rounded toward zero,
- rounding to the nearest integer, with 0.5 being rounded away from zero,
- rounding toward zero, and
- rounding away from zero.

[0028] It should be noted that, if the first rounding mode is selected, the shift right operation with rounding is entirely equivalent to the conventional shift right operation. Depending on the particular rounding mode, the number *s* of bits shifted, and the sign of the shifted operand, a term is added to the operand to be shifted, as shown at a block 202. Table 1 shows the term that is added in each case, assuming that *s* is greater than zero. If s is equal to zero, *i.e.*, if no shifting is performed, then no term is added to the operand, and no rounding is applied. The integer rounding mode values listed in Table 1 below are provided for purposes of illustration only.

TABLE 1

Integer rounding mode value	Rounding mode	Term to be added, positive operand	Term to be added, negative operand
000	To negative infinity	0	0
001	To nearest, half to negative infinity	$2^{s-1} - 1$	$2^{s-1} - 1$
010	To zero	0	$2^{s} - 1$
011	To nearest, half to zero	$2^{s-1} - 1$	$2^{s} - 1$

100	To infinity	$2^s - 1$	$2^s - 1$
101	To nearest, half to positive infinity	$2^s - 1$	$2^s - 1$
110	Away from zero	$2^s - 1$	0
111	To nearest, half away from zero	$2^s - 1$	$2^{s-1} - 1$

[0029] After the term is added (if applicable), the operand is shifted *s* bits to the right at a block 204.

[0030] As an example, if r10, interpreted as a half-word signed-integer vector, represents the vector (-4, -3, 3, 4) and the integer rounding mode value is 011, *i.e.*, round to the nearest integer, with 0.5 being rounded toward zero, then the result of shifting right by one bit with rounding is (-2, -1, 1, 2). This result is obtained by first adding the term vector (1, 1, 0, 0) to r10 as specified by the integer rounding mode value, resulting in (-3, -2, 3, 4) and then shifting right by one bit, so as to divide by two with rounding toward negative infinity.

[0031] Similarly, the result of a multiplication operation has twice as many bits as the operands. Dropping the least significant half of the bits corresponds to dividing the result by a power of two. For example, multiplying two eight-bit operands results in a product having sixteen bits. Dropping the least significant eight bits corresponds to dividing the result by 2^8, or 64. This operation can also be performed with any of the integer rounding modes described above in connection with Table 1.

[0032] Figure 3 illustrates an example mode of operation 300 of the processor 10 of Figure 1 for performing a rounded average of four unsigned-byte vectors, according to another embodiment of the invention. This operation takes the syntax

[if *rguard*] avg4_bu rsrc1 rsrc2 rsrc3 rsrc4 rdest

where rsrc1, rsrc2, rsrc3, rsrc4, and rdest are interpreted as unsigned-byte vectors. In a particular type of VLIW processor, this operation issues from a functional unit of type *superdspalu*, uses issue slots 3 and 4, and has a latency of two cycles. The operation takes the rounded average of the four unsigned-byte vectors rsrc1, rsrc2, rsrc3, and rsrc4 by adding the vectors at a block 302 and dividing by four, *i.e.*, right-shifting by two bits, at a block 304. Rounded averaging is implemented by adding two prior to the right-shift operation, as shown at a block 306. Although this added quantity is shifted out in connection with block 304, the rounding hardware perceives it as a value of one-half, and applies rounding accordingly. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. The rounded averaging operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

By way of example, the operation

avg4_bu r10 r20 r30 r40 r50

where

r10 = 0xff_ff_ff_02_63_1e_00_d1

r20 = 00_01_ff_03_02_00_01_d2

r30 = 0xff_ff_ff_96_01_1e_00_d1

r40 = 00_01_ff_01_01_01_01_d2

produces the result

r50 = 0x80_80_ff_27_1a_0f_01_d2.

[0033] Another operation that can be performed is a rounded average of two unsigned-byte vectors, as shown by the example mode of operation 400 of Figure 4. This operation takes the syntax

[if *rguard*] avg_bu rsrc1 rsrc2 rdest

where rsrc1, rsrc2, and rdest are interpreted as unsigned-byte vectors. In a particular type of VLIW processor, this operation issues from a functional unit of type *dspalu*, uses issue slots 3 and 4, and has a latency of two cycles. The operation takes the rounded average of the two unsigned-byte vectors rsrc1 and rsrc2 by adding the vectors at a block 402 and dividing by two, *i.e.*, right-shifting by one bit, at a block 404. Rounded averaging is implemented by adding one prior to the right-shift operation, as shown at a block 406. Although this added quantity is shifted out in connection with block 404, the rounding hardware perceives it as a value of one-half, and applies rounding accordingly. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. The rounded averaging operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0034] As an example, the operation

avg_bu r10 r20 r30

where

r10 = 00_ff_a0_fa_ff_e6_78_0a

r20 = 00_ff_a0_ff_f0_f0_82_14

produces the result

r30 = 00_ff_a0_fd_f8_eb_7d_0f.

[0035] According to another implementation of the invention, the processor is configured to perform a non-saturating fixed-point fractional multiplication operation with rounding of a signed- and an unsigned-byte vector. Figure 5 depicts an example mode of operation 500 of the processor 10 of Figure 1 for performing an operation of this type. This operation takes the syntax

[if *rguard*] mspmul_bsus rsrc1 rsrc2 rdest

where rsrc1 and rdest are interpreted as signed-byte vectors and rsrc2 is an unsigned-byte vector. In a particular type of VLIW processor, this operation issues from a functional unit of type *mul*, uses issue slots 2 and 3, and has a latency of three cycles. For each vector element, this operation performs a non-saturating fixed-point fractional two-quadrant multiplication operation with rounding of a signed-byte vector and an unsigned-byte vector, as shown at a block 502.

[0036] The upper eight bits of each 16-bit product are stored in the corresponding element in the rdest register. As described above, dropping the least significant half of the bits corresponds to dividing the result by a power of two. In this case, the lower eight bits are dropped by right-shifting by eight bits, as shown at a block 504. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term, which may be zero, is added to the product at a block 506 prior to right-shifting. The fractional multiplication operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place. For example, the operation

mspmul_bsus r10 r20 r30

where

r10 = 0xfe_ff_01_80_f0_10_f0_10

r20 = 0xff_ff_ff_ff_02_02_10_08

and the integer rounding mode value is 000 (*i.e.*, round toward negative infinity) produces the result

r30 = 0xfe_ff_00_80_ff_00_ff_00.

[0037] Still another type of operation that can be performed is a non-saturating fixed-point fractional multiplication operation with rounding of two unsigned-byte vectors. Figure 6 illustrates an example mode of operation 600 of the processor 10 of Figure 1 for performing such an operation. This operation takes the syntax

[if *rguard*] mspmul_bu rsrc1 rsrc2 rdest

where rsrc1, rsrc2, and rdest are interpreted as unsigned-byte vectors. In a particular type of VLIW processor, this operation issues from a functional unit of type *mul*, uses issue slots 2 and 3, and has a latency of three cycles. For each vector element, this operation performs a non-saturating fixed-point fractional one-quadrant multiplication operation with rounding of two unsigned-byte vectors, as shown at a block 602.

[0038] The upper eight bits of each 16-bit product are rounded and stored in the corresponding element in the rdest register. As described above, dropping the least significant half of the bits corresponds to dividing the result by a power of two. In this case, the lower eight bits are dropped by right-shifting by eight bits, as shown at a block 604. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the product at a block 606 prior to right-shifting. The fractional multiplication operation may make use of an optional guard register, as

specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0039] For example, the operation

mspmul_bu r10 r20 r30

where

r10 = 0x10_02_02_80_10_80_ff

r20 = 00_0f_80_ff_02_10_10_01

and the integer rounding mode value is 010 (*i.e.*, round toward zero) produces the result

r30 = 00_00_01_01_01_01_08_00.

[0040] According to still another embodiment, the processor is also configured to perform a non-saturating fixed-point fractional multiplication operation with rounding of two signed half-word vectors, for example, using a mode of operation 700 of Figure 7. This operation takes the syntax

[if *rguard*] mspmul_hs rsrc1 rsrc2 rdest

where rsrc1, rsrc2, and rdest are interpreted as signed half-word vectors. In a particular type of VLIW processor, this operation issues from a functional unit of type *mul*, uses issue slots 2 and 3, and has a latency of three cycles. For each vector element, this operation performs a non-saturating fixed-point fractional four-quadrant multiplication operation with rounding of two signed half-word vectors, as shown at a block 702.

[0041] The upper sixteen bits of each 32-bit product are rounded and stored in the corresponding element in the rdest register. Dropping the least significant half of the bits corresponds to dividing the result by a power of two. In this case, the lower sixteen bits are dropped by right-shifting by sixteen bits, as shown at a block 704. The type of rounding applied depends on the value of an integer rounding mode value, for example,

as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the product at a block 706 prior to right-shifting. The fractional multiplication operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0042] For example, the operation

mspmul_hs r10 r20 r30

where

r10 = 0x4000_8000_4000_7fff

r20 = 0xfff3_ffff_0002_0001

and the integer rounding mode value is 000 (*i.e.*, round toward negative infinity) produces the result

r30 = 0xffff_0001_0001_0000.

[0043] In yet another embodiment, the processor is configured to perform a non-saturating fixed-point fractional multiplication operation with rounding of a signed half-word vector and an unsigned half-word vector, for example, using a mode of operation 800 of Figure 8. This operation takes the syntax

[if *rguard*] mspmul_hsus rsrc1 rsrc2 rdest

where rsrc1 and rdest are interpreted as signed half-word vectors and rsrc2 is an unsigned half-word vector. In a particular type of VLIW processor, this operation issues from a functional unit of type *mul*, uses issue slots 2 and 3, and has a latency of three cycles. For each vector element, the operation performs a non-saturating fixed-point fractional two-quadrant multiplication operation with rounding of a signed half-word vector and an unsigned half-word vector, as shown at a block 802. The lower sixteen bits of each 32-

bit product are rounded and stored in the corresponding element in the rdest register by right-shifting by sixteen bits, as shown at a block 804. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the product at a block 806 prior to right-shifting. The fractional multiplication operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0044] For example, the operation

mspmul_hsus r10 r20 r30

where

r10 = 0x8000_ffff_0002_0100

r20 = 0xffff_4000_ffff_0040

and the integer rounding mode value is 000 (*i.e.*, round toward negative infinity) produces the result

r30 = 0x8000_ffff_0001_0000.

[0045] According to still another embodiment, the processor is also configured to perform a non-saturating fixed-point fractional multiplication operation with rounding of two unsigned half-word vectors, for example, using a mode of operation 900 of Figure 9. This operation takes the syntax

[if *rguard*] mspmul_hu rsrc1 rsrc2 rdest

where rsrc1, rsrc2, and rdest are interpreted as unsigned half-word vectors. In a particular type of VLIW processor, this operation issues from a functional unit of type *mul*, uses issue slots 2 and 3, and has a latency of three cycles. For each vector element,

this operation performs a non-saturating fixed-point fractional one-quadrant multiplication operation with rounding of two unsigned half-word vectors, as shown at a block 902.

[0046] The upper sixteen bits of each 32-bit product are rounded and stored in the corresponding element in the rdest register. As described above, dropping the least significant half of the bits corresponds to dividing the result by a power of two. In this case, the lower sixteen bits are dropped by right-shifting by sixteen bits, as shown at a block 904. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the product at a block 906 prior to right-shifting. The fractional multiplication operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0047] For example, the operation

mspmul_hu r10 r20 r30

where

r10 = 0xfffe_ffff_ffff_0100

r20 = 0x01_8000_ffff_0040

and the integer rounding mode value is 010 (*i.e.*, round toward zero) produces the result

r30 = 0000_7fff_fff3_0000.

[0048] In yet another embodiment, the processor is also configured to perform a non-saturating fixed-point fractional multiplication operation with rounding of two signed word vectors, for example, using a mode of operation 1000 of Figure 10. This operation takes the syntax

[if *rguard*] mspmul_ws rsrc1 rsrc2 rdest

where rsrc1, rsrc2, and rdest are interpreted as signed word vectors. In a particular type of VLIW processor, this operation issues from a functional unit of type *mul*, uses issue slots 2 and 3, and has a latency of three cycles. For each vector element, this operation performs a non-saturating fixed-point fractional four-quadrant multiplication operation with rounding of two signed word vectors, as shown at a block 1002.

[0049] The upper thirty-two bits of each 64-bit product are rounded and stored in the corresponding element in the rdest register. Dropping the least significant half of the bits corresponds to dividing the result by a power of two. In this case, the lower thirty-two bits are dropped by right-shifting by thirty-two bits, as shown at a block 1004. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the product at a block 1006 prior to right-shifting. The fractional multiplication operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0050] For example, the operation

mspmul_ws r10 r20 r30

where

r10 = 0x004000_ffff_8000

r20 = 0x7fffffffe_ffffff

and the integer rounding mode value is 000 (*i.e.*, round toward negative infinity) produces the result

r30 = 0x7ffffffe_00000000.

[0051] The processor can also be configured to perform a non-saturating fixed-point fractional multiplication operation with rounding of a signed word vector and an unsigned word vector, for example, using a mode of operation 1100 of Figure 11. This operation takes the syntax

[if *rguard*] mspmul_wsus rsrc1 rsrc2 rdest

where rsrc1 and rdest are interpreted as signed word vectors and rsrc2 is an unsigned word vector. In a particular type of VLIW processor, this operation issues from a functional unit of type *mul*, uses issue slots 2 and 3, and has a latency of three cycles. For each vector element, the operation performs a non-saturating fixed-point fractional two-quadrant multiplication operation with rounding of a signed word vector and an unsigned word vector, as shown at a block 1102.

[0052] The lower sixteen bits of each 32-bit product are rounded and stored in the corresponding element in the rdest register by sixteen bits, as shown at a block 1104. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the product at a block 1106 prior to right-shifting. The fractional multiplication operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0053] For example, the operation

mspmul_wsus r10 r20 r30

where

r10 = 0x80000000

r20 = 0x7fffffff

and the integer rounding mode value is 000 (*i.e.*, round toward negative infinity) produces the result

r30 = 0xc0000000.

[0054] As another example, assuming

r10 = 0x7fffffff_80000000

r20 = 0x80000000_7fffffff

the same operation produces the result

r30 = 0xffffffff_ffffffff.

[0055] The processor can also be configured to perform a non-saturating fixed-point fractional multiplication operation with rounding of two unsigned word vectors, for example, using a mode of operation 1200 of Figure 12. This operation takes the syntax

[if *rguard*] mspmul_wu rsrc1 rsrc2 rdest

where rsrc1, rsrc2, and rdest are interpreted as unsigned word vectors. In a particular type of VLIW processor, this operation issues from a functional unit of type *mul*, uses issue slots 2 and 3, and has a latency of three cycles. For each vector element, this operation performs a non-saturating fixed-point fractional one-quadrant multiplication operation with rounding of two unsigned word vectors, as shown at a block 1202. The upper thirty-two bits of each 64-bit product are rounded and stored in the corresponding element in the rdest register.

[0056] As described above, dropping the least significant half of the bits corresponds to dividing the result by a power of two. In this case, the lower thirty-two bits are dropped by right-shifting by thirty-two bits, as shown at a block 1204. The type of rounding applied depends on the value of an integer rounding mode value, for

example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the product at a block 1206 prior to right-shifting. The fractional multiplication operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0057] For example, the operation

mspmul_wu r10 r20 r30

where

r10 = 0x80000000_80000001

r20 = 0x80000000_ffffffff

and the integer rounding mode value is 010 (*i.e.*, round toward zero) produces the result

r30 = 0x40000000_80000000.

[0058] In another particular embodiment of the invention, the processor is further configured to perform an arithmetic right-shift of a signed-byte vector by an immediate shift amount with rounding. This operation can be performed, for example, using a mode of operation 1300 of Figure 13. The operation takes the syntax

[if *rguard*] sriround_bs (*s*) rsrc1 rdest

where rsrc1 and rdest are interpreted as signed byte vectors and *s* is an immediate argument in the range of 0 to 7. In a particular type of VLIW processor, this operation issues from a functional unit of type *shift_round*, uses issue slots 1 and 2, and has a latency of two cycles. For each destination element, the operation performs an arithmetic, *i.e.*, sign extending right-shift operation on the corresponding element of rsrc1 depending on the value of the immediate argument *s*, as shown at a block 1302. The type

of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the operand at a block 1304 before shifting is performed. The operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0059] By way of example, the operation

sriround_bs (2) r10 r20

where

r10 = 0x04_03_02_01_00_ff_fe_fd

and the integer rounding mode value is 011 (*i.e.*, round to the nearest integer, with half being rounded to zero) produces the result

r20 = 0x01_01_00_00_00_00_00_ff.

On the other hand, the same operation with the integer rounding mode value set to 101 (*i.e.*, round to the nearest integer, with half being rounded toward positive infinity) produces the result

r20 = 0x01_01_01_00_00_00_00_ff.

With the integer rounding mode value set to 001 (*i.e.*, round to the nearest integer, with half being rounded toward negative infinity), the result is instead

r20 = 0x01_01_00_00_00_00_ff_ff.

[0060] In still another embodiment of the invention, the processor is further configured to perform a logical right-shift of an unsigned-byte vector by an immediate shift amount with rounding. This operation can be performed, for example, using a mode of operation 1400 of Figure 14. The operation takes the syntax

[if *rguard*] sriround_bu (*s*) rsrc1 rdest

where rsrc1 and rdest are interpreted as unsigned byte vectors and *s* is an immediate argument in the range of 0 to 7. In a particular type of VLIW processor, this operation issues from a functional unit of type *shift_round*, uses issue slots 1 and 2, and has a latency of two cycles. For each destination element, the operation performs a logical, *i.e.*, zero extending right-shift, operation on the corresponding element of rsrc1 depending on the value of the immediate argument *s*, as shown at a block 1402. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the operand at a block 1404 before shifting is performed. The operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0061] As an example, the operation

sriround_bu (2) r10 r20

where

r10 = 0x07_06_05_04_03_02_01_00

and the integer rounding mode value is 011 (*i.e.*, round to the nearest integer, with half being rounded to zero) produces the result

r20 = 0x02_01_01_01_01_00_00_00.

On the other hand, the same operation with the integer rounding mode value set to 101 (*i.e.*, round to the nearest integer, with half being rounded toward positive infinity) produces the result

r20 = 0x02_02_01_01_01_01_00_00.

With the integer rounding mode value set to 001 (*i.e.*, round to the nearest integer, with half being rounded toward negative infinity), the result is instead

r20 = 0x02_01_01_01_01_00_00_00.

[0062] Another operation that can be performed is an arithmetic right-shift of a signed double word by an immediate shift amount with rounding. This operation can be performed, for example, using a mode of operation 1500 of Figure 15. The operation takes the syntax

[if *rguard*] sriround_ds (*s*) rsrc1 rdest

where rsrc1 and rdest are interpreted as signed double word vectors and *s* is an immediate argument in the range of 0 to 63. In a particular type of VLIW processor, this operation issues from a functional unit of type *shift_round*, uses issue slots 1 and 2, and has a latency of two cycles. For each destination element, the operation performs an arithmetic, *i.e.*, sign extending right-shift operation on the corresponding element of rsrc1 depending on the value of the immediate argument *s*, as shown at a block 1502. The type of rounding applied depends on the value of an integer rounding mode value, for example, as shown above in Table 1. Depending on the rounding mode, a rounding term is added to the operand at a block 1504 before shifting is performed. The operation may make use of an optional guard register, as specified in *rguard*. If a guard register is present and has a value of TRUE, the operation is performed. If the guard register has a value of FALSE, no operation takes place.

[0063] By way of example, the operation

sriround_ds (2) r10 r20

where

r10 = 0xffffffff_fffffffd

and the integer rounding mode value is 011 (*i.e.*, round to the nearest integer, with half being rounded to zero) produces the result

r20 = 0xffffffff_ffffffff.

The same result is obtained with the integer rounding mode value set to 101 (*i.e.*, round to the nearest integer, with half being rounded toward positive infinity) or 001. With the integer rounding mode value set to 010 (*i.e.*, round toward zero), the result is instead

r20 = 0x0_0.

[0064] According to another embodiment of the invention, any of the integer rounding operations described above in connection with Figure 2-15 can be implemented in a system 1600 of Figure 16. The system 1600 includes a processor 1602, such as the processor 10 of Figure 1, and a memory 1604.

[0065] Various methods for performing integer rounding operations in a computer processor have been described. Besides conventional rounding operations, *i.e.*, rounding down toward negative infinity in all cases, alternative rounding modes can be specified. Depending in part on the particular rounding mode used and on the sign of the value to be rounded, a rounding term is added to the value to be rounded before right-shifting is performed. Adding the rounding term ensures that the desired result is obtained.

[0066] It is to be understood that, even though numerous characteristics and advantages of various embodiments of the invention have been set forth in the foregoing description, together with details of the structure and function of various embodiments of the invention, this disclosure is illustrative only, and changes may be made within the

principles of the invention to the full extent indicated by the broad general meaning of the terms in which the appended claims are expressed.